Linda: A paradigm for parallel programming

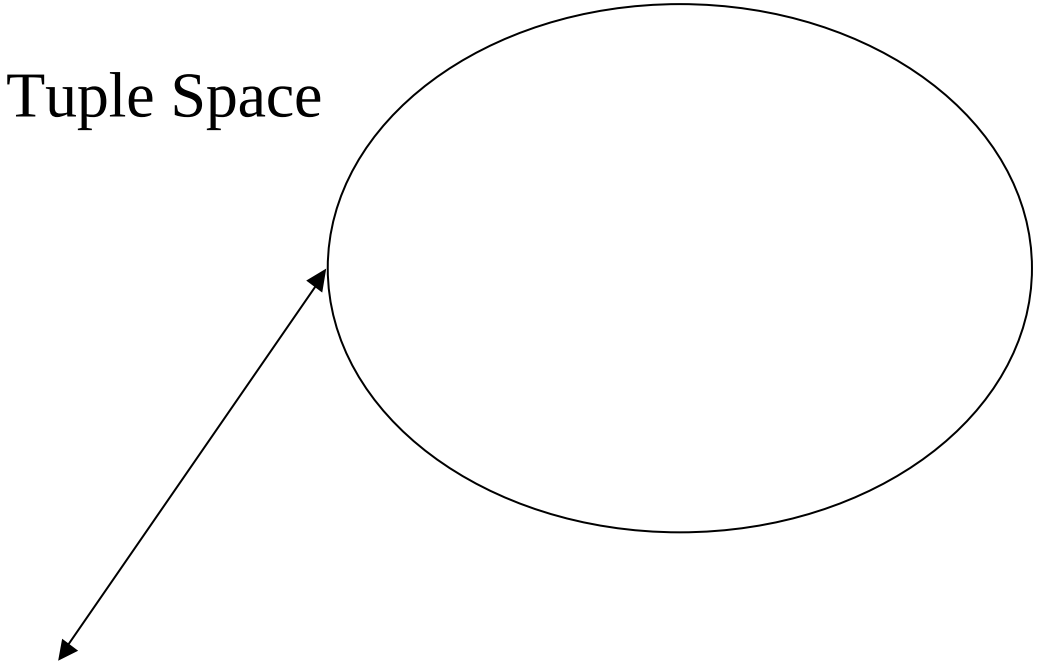
Plan for MVP 13

1. What is Linda?
2. Conceptual classes for parallellism:
   - Result, Agenda, and Specialist –Parallellism
3. Programming methods for parallellism:
   - Live data structures, Distributed data structures, Message passing
4. A simple embedding of Linda in Java
5. A case study: Finding prime numbers

What is Linda?

- A Linda system consists of a set of processes (or threads) and a synchronized *Virtual Shared Memory* called *Tuple Space*
- Objects (tuples) enter/leave Tuple Space using the *Linda primitives*: in,out,read,eval
- The primitives are *orthogonal* to the applied programming language – i.e. Linda can be embedded in any programming language.

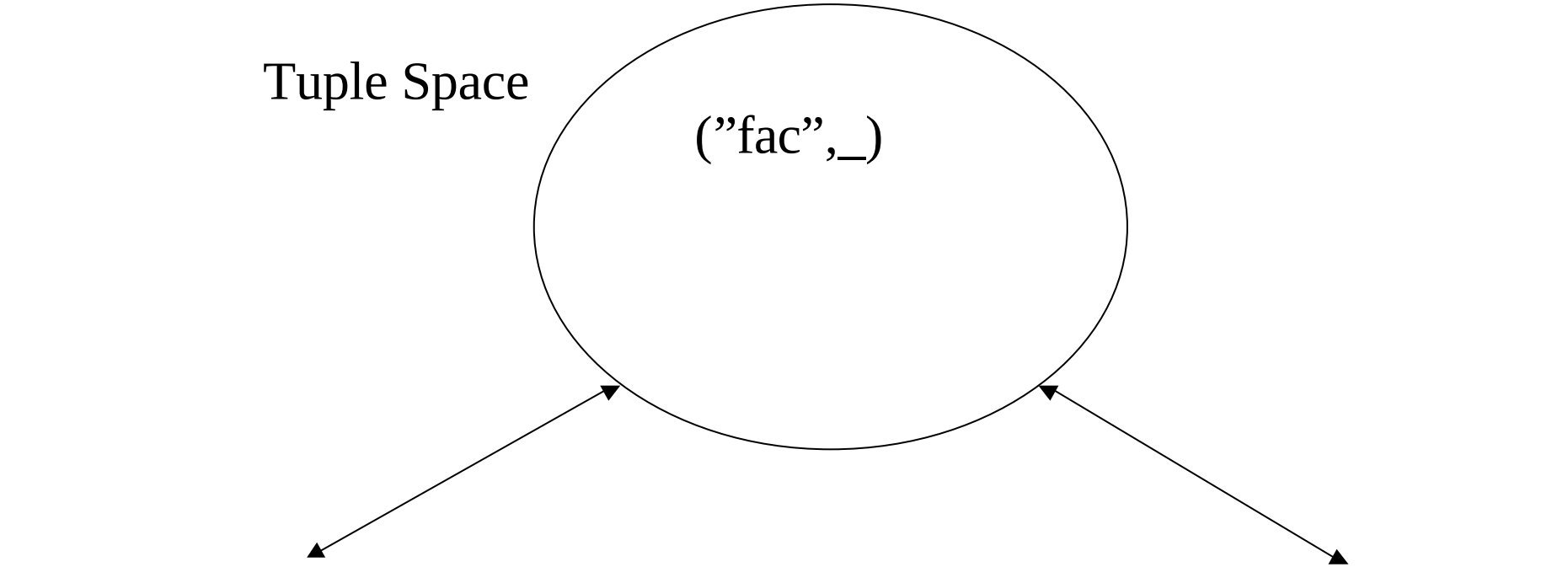
Illustration of tuple operations



P1:
eval(”fac”,f(3));
in(”fac”,?i);
println(i);

```
int f(int n){
if (n==0) return(1)
else return(n*f(n-1))
}
```

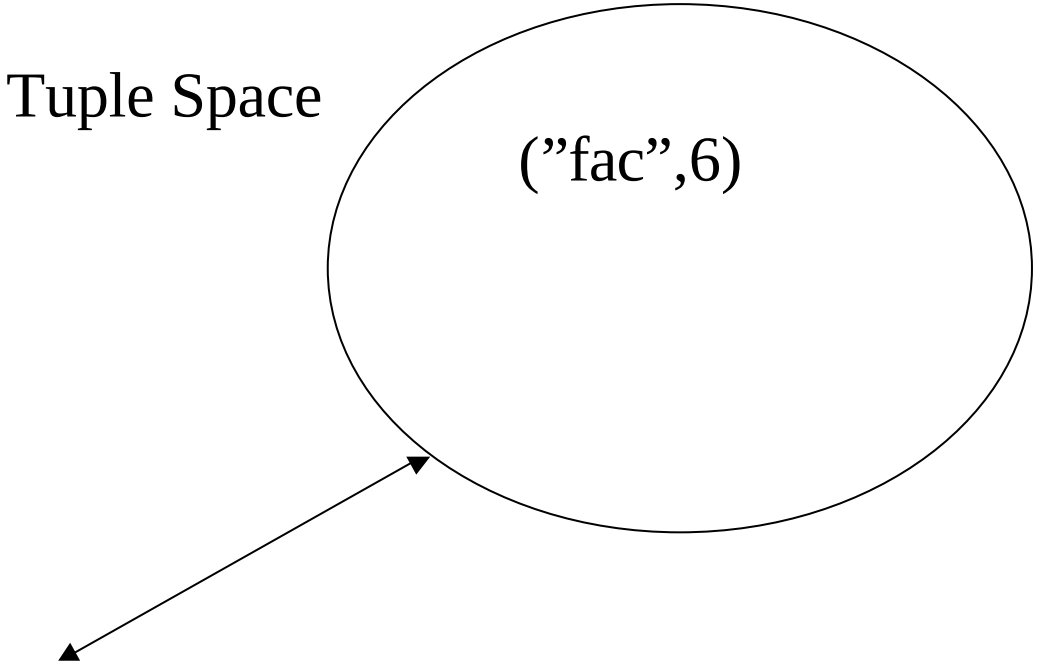
Illustration of tuple operations



P1:
eval(”fac”,f(3));
in(”fac”,?i);
println(i);

P2:
int f(int n){
if (n==0) return(1)
else return(n*f(n-1))
}

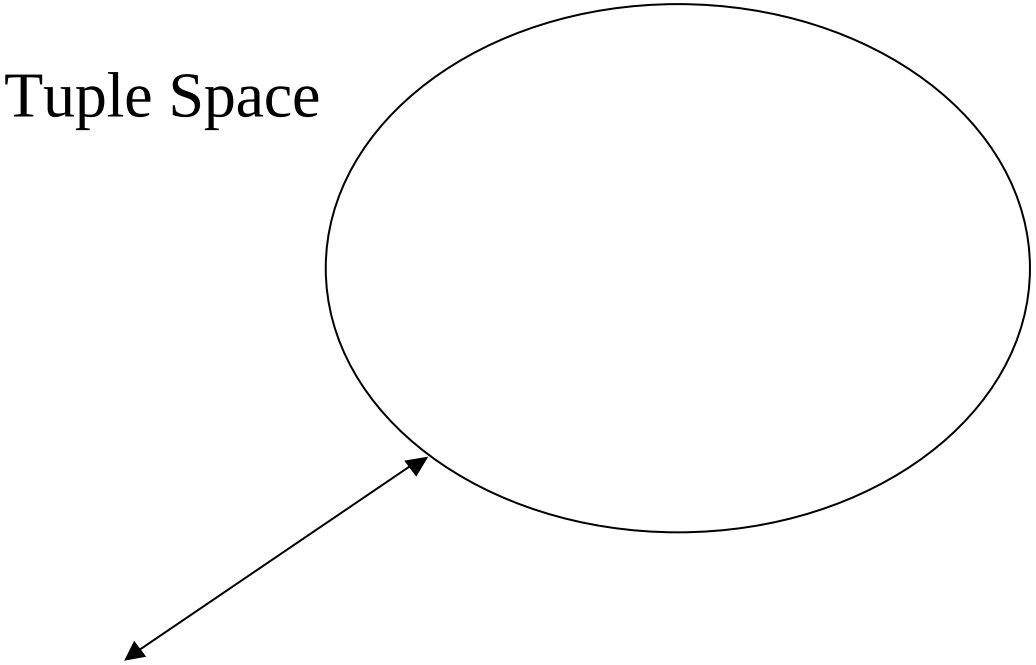
Illustration of tuple operations



P1:
eval(”fac”,f(3));
in(”fac”,?i);
println(i);

```
int f(int n){
if (n==0) return(1)
else return(n*f(n-1))
}
```

Illustration of tuple operations



P1:
eval(”fac”,f(3));
in(”fac”,?i);
println(i); /* i == 6 */

```
int f(int n){
if (n==0) return(1)
else return(n*f(n-1))
}
```

Linda tuple operations

- tuple::=(<value>,…,<value>)
- template::=(<value>|?<variable>,…, <value>|?<variable>)
- out(tuple t):
  - Enter tuple t into Tuple Space
- read(template t):
  - Bind a copy of a matching tuple to t (or block if no mathcing tuple exists)
- in(template t):
  - Works as ‘read’ and removes the mathcing tuple from Tuple Space

Linda tuple operations (cntd)

- eval(tuple t):
  - Enter (active tuple) t into Tuple Space, and spawn a process for each function value to be computed
  - t is not made available for matching until all computations have terminated (and t has become ‘passive’)

Matching rules

- Tuple t=(t1,…tN) *matches* template tp=(tp1,…tpM) iff:
  - N==M
  - For all i:
    - The types of ti and tpi conform
    - ti==tpi, if tpi is a value
- If t and tp match in a read/in operation, the operation *binds* the values ti to tpi if tpi is a variable.

Typical data structures in Tuple Space

- Semaphores:
  - P(sem): in(“sem”)
  - V(sem): out(“sem”)

  } Can easily be extended to a counting semaphore

- Bags:
  - out(“Id”,<value>)
  - in(“Id”,<?variable>)

  } No unique naming, i.e. *some* matching tuple is remove from Tuple Space

Typical Data Structures in Tuple Space (cntd)

- Distributed arrays:
  - Enter A(i,j) into Tuple Space: out(“A”,i,j,<value>)
  - Read A(i,j) from Tuple Space: rd(“A”,i,j,?<variable>)

Typical Data Structures in Tuple Space (cntd)

- Distributed streams:
  - elements: (“strm”,index,element value)
  - Remove first element:
    - in(“strm”,”head”,?index)
    - out(“strm”,”head”,index+1)

    } Update head (if several readers)
    - in(“strm”,index,?element)

    } Remove element
  - Add new element:
    - in(“strm”,”tail”,?index)
    - out(“strm”,”tail”,index+1)

    } Update tail (if several writers)
    - in(“strm”,index,?element)

    } Add element

Conceptual classes for parallellism (result,agenda,message passing)

- Result parallellism:
  - Basic idea: Each component of the computation result is computed by a “worker” (thread). A “worker” may have to refer to other (sub) results.
  - House building example: Each part of the house is made by *one* specific worker.
  - Simple example: parallel array addition/multiplication.

Conceptual classes (cntd)

- Specialist parallellism:
  - Basic idea: Each *type* of component is computed by a ”worker” (thread), i.e. a thread may produce many similar parts of the result (and refer to other result parts if necessary).
  - House building example: All doors are made by a specific worker.
  - Simple example: Transport simulation – often organised into static networks.

Conceptual classes (cntd)

- Agenda parallellism:
  - Basic idea: The result is obtained through a set of (sometimes sequential) subtasks. All "workers" (threads) participate in *several* subtasks.
  - House building example: foundation ⟶ walls ⟶ roof ⟶ (plumbing || wiring) ⟶ …
  - Simple example: Decomposition of solving matrix equations (AX=B: (1)LY=B, (2)UX=B).

Programming Methods
(Live and Distributed Data Structures, Message Passing)

- Message Passing:
  - Explicit communication between processes
  - Data and results are encapsulated in processes
- Distributed Data Structures:
  - Separation of data and processes
  - Communication via shared data objects

Programming Methods (cntd)

- Live Data Structures:
  - The resulting data structure determines the program structure
  - Processes are encapsulated in result data.
  - Implicit communication via results
  - Results are visible, when the encapsulated processes have terminated.

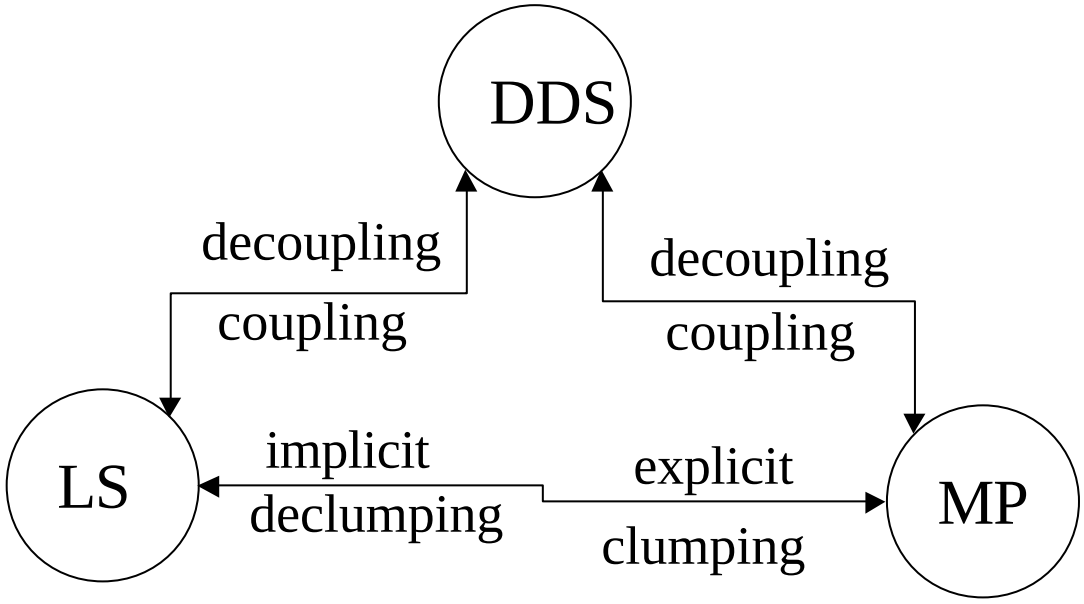
Classes vs Methods

- Live Data Structures ⟷ Result
- Message Passing ⟷ Specialist
- Distributed Data Structures ⟷ Agenda

Method Selection Strategy

1. Start with a ’natural’ class.
2. Make a program using the corresponding method.
3. If necessary (i.e. performance is poor), then transform the program using an alternative method.

Method transformation



- Decoupling of data and processes
- Explicit communication
- Clumping of set of subresults into larger groups

Taxonomi

	#Skills	#Tasks	Granularity	Coordination
Result	one	one	fine	Tight
Specialization	one	many	medium	Moderate
Agent	many	many	flexible	Loose

A simple Linda-embedding in JAVA (by Carsten Stiborg)

- No eval implementation. Explicit threads.
- TupleSpace ts = new TupleSpace()
- ts.out(new Tuple(new Object[]{"primes",new Integer(me), new Boolean(ok)}));
- Tuple in = ts.read(new Tuple(new Object[] {"primes",new Integer(i),new Boolean(false).getClass()})); ok=((Boolean)in.getActual(2)).booleanValue();

Case study: Prime numbers

- Problem: Find all prime numbers in [2..N]
- Result parallellism:
  - Observe that p is a prime iff p mod j !=0 for all prime numbers j<=sqrt(p)
  - Start a thread for each potential prime number:
    - For i in [2..N] eval(“primes”,i,is_prime(i))
  - Read results from Tuple Space:
    - For i in [2..N] read(“primes”,i,?ok)

Result parallellism (cntd)

- is_prime(me):
  - for i in [2..sqrt(me)] (loop test: i*i<=me)
    {read("primes",i,?ok);
    if (ok && (me % i==0)) return false;
    };
    return true
- A java version:
  - http://www.cs.auc.dk/~ask/Undervisning/MVP/testprime.txt

Problems with result solution

- Many threads (overhead, memory)
- Small granularity (i.e. each thread computes only a small amount of work)
- Alternative using agent parallellism:
  - Group [2..N] into intervals (of size grainsize)
  - A fixed number of worker threads compute the primes of the intervals and report results back to a master thread

Agent solution (cntd)

- Additional optimization:
  - Each worker keeps the applied primes (and their squares) in a local array
- Worker main loop:
  -

```
in(”next task”,?start);
out(”next task”,start+grainsize);
…compute primes in [start…start+grainsize[
```

- The number of possible concurrent workers grows quadratic with the largest prime found so far

Specialist parallellism: Eratosthenes' sieve

- One specialist worker thread for each prime
- Main loop of worker(p,nextp):
  - in(”stream”,p,i_index++,?number);
    if (!number){//end of stream
      out(”stream”,next,o_index,number); //write end of next stream};
    if (number % p) out(”prime”,next,o_index++,number);
- Master collects results and starts new specialist
- Problem: Low degree of concurrency

Conclusions

- Linda was developed in mid 80’s
- Linda may be implemented *efficiently*
- Commercial applications within scientific computing
- SUN and IBM have released JAVA based versions (JavaSpaces, Tspaces)